**Graphs 3**

**Edge-List Representation**

For large numbers of vertices, the adjacency matrix representation is costly in both memory space O(V2) and run-time O(V3). In contrast, the *adjacency list* (or *edge-list*) representation stores vertices with their edges, which requires O(V\*E) space. If the given graph is sparse, i.e., if E<V, you should probably use the edge-list representation. The edge-list representation does not solve the all-pairs reachability problem, but for many purposes, we don’t need to.

In the edge-list representation, each vertex maintains a list of references to its neighboring vertices. Given the “logical representation” of the graph on the left, its “edge-list representation” is on the right:

|  |  |
| --- | --- |
| A | [C] |
| B | [A] |
| C | [C D] |
| D | [C A] |

The edge-list data structure therefore is more complicated than the adjacency-matrix data structure. Generally, from now on, we will build complicated graphs using Vertexes, just as before we built complicated linked lists using ListNodes and complicated trees using TreeNodes.

We need a Vertex class. Each Vertex object will have fields to store its own name and a set of its neighboring vertices. For example, the third Vertex object above will store both "C" and the set  
[C D]. We assume that names (and therefore Vertex objects) are unique. We use a set to store the adjacencies because sets don't allow duplicates. Because we are using a HashSet, we need to override equals(Object) and hashCode to make the set behave properly. Both those methods need to work on the field that stores the name. What methods do you need in the Vertex class? List them:

class Vertex implements VertexInterface, Comparable<Vertex>

{  
 private final String name;

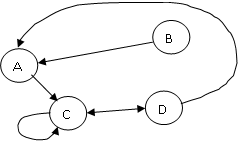
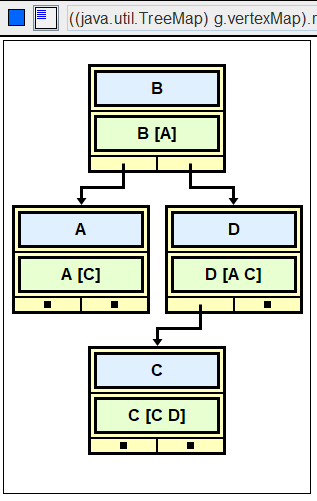
private HashSet<Vertex> adjacencies;

Since a graph is just an arrangement of Vertexes, we don’t actually need a graph class. Just as linked lists are arrangements of ListNodes, or trees are arrangements of TreeNodes, we could work directly with the Vertexes. However, it is convenient to encapsulate all the fields and methods in a graph class. Let’s call it the AdjList class. There is no standard structure for either Vertex or graph. Every programmer makes classes that are tailored to the problem. We chose this one for our graph:

public class AdjList implements AdjListInterface  
{  
 private Map<String, Vertex> vertexMap = new TreeMap<String, Vertex>();

The BST data structure of every TreeMap means the vertices can be printed in alphabetical order. Each node in the TreeMap maps a name (the blue box) to its Vertex (the green box). Each Vertex stores its own name and a set of adjacencies (or neighbors). (Yes, each vertex's name is stored in two different places.) The picture and its (slightly different) edge-list representation produces the BST structure of the vertexMap as shown by jGrasp:

|  |  |
| --- | --- |
| A | [C] |
| B | [A] |
| D | [C A] |
| C | [C D] |



For this lab and the next, we are going to hard-code the vertices and the edges in the driver. List the methods you need in AdjList:

public class AdjList implements AdjListInterface

{  
 private Map<String, Vertex> vertexMap = new TreeMap<String, Vertex>();

/\* constructor is not needed because of the instantiation above \*/

**addEdge**

The method addEdge does not need any special cases if make sure that all the vertices are in the graph first. A programmer's solution is to write a precondition to the addEdge method:

/\*  
 precondition: both Vertexes, source and target, are already   
 stored in the graph

postcondition: addEdge should work in O(log n)  
 \*/  
 public void addEdge(String source, String target)

And here is a warning: in addEdge, students love to add a brand-new target Vertex to the graph. That won’t work. You must make the edge connect the source and target Vertexes that have already been stored in the graph.

**Assignment**: Implement Vertex and AdjList. The AdjList\_3\_Driver first tests the methods in the Vertex class and then the methods in the AdjList graph.

**Sample Run:** (AdjList\_3\_Driver.java using AdjList)

Edge List Representation!   
Test the Vertex class  
Add some vertices and adjacencies.  
get the name and adjacencies:  
 zeta [theta [zeta theta]]  
get the name and adjacencies:  
 theta [zeta [theta], theta [zeta theta]]  
toString() shows the names plus the name of the neighbor(s):   
 zeta [theta]  
 theta [zeta theta]  
  
Test the AdjList class  
Adding some vertices and edges.  
set of vertices in the graph: [A [C], B [A], R [], C [R C D], D [A C]]  
map string to vertex: {A=A [C], B=B [A], C=C [R C D], D=D [A C], R=R []}  
get a vertex by name "B": B [A]  
  
toString the whole graph:  
A [C]  
B [A]  
C [R C D]  
D [A C]  
R []

**Things to check**

1. Don't add the edges directly to the list. Edges go onto the stack before they go onto the list. Follow the algorithm as given!
2. Don't use a TreeSet to fix the order of the vertices, because the TreeSet rearranges the HashSet.
3. Spell "hashCode" correctly. Override it to work on names only.
4. Override equals(Object) to work on the name only:

public boolean equals(Object obj)  
 {  
 if( obj instanceof Vertex)  
 return name.equals(((Vertex)obj).name);  
 return false;   
 }